**what is a git commit ?**

A commit is the group of -

1.Saved changes to the Git repository

2.This impacts history

3.Uniquely identified by a SHA-1 hash

**What is a git branch?**

Branches can be thought of as a timeline with commits. By default, the master branch is the main, primary branch that we usually work with. HEAD is a pointer to the last commit on the current branch. Remote is simply a refine, or a pointer, to a related repository somewhere that's not local and that could be within local network or enterprise network or somewhere out on the internet. Some examples of places that might be hosting remote repositories would include GitHub, Gitlab, Bitbucket etc.

**How to check if git is installed and how to get help on various git commands?**

On the terminal/cmd using 'git version' command; it will show the installed git's version. Using command 'git help'; we can display the list of commands supported by the git. Using 'git help <command-Name>' we can get the detailed information about specific git command; on windows it will be displayed in the web-browser but on Mac/Linux it will be displayed on the terminal itself. Command 'git help -a' gives a list of subcommands.

**How to get the current status of local repository?**

Using 'git status' command in the repository working directory, when this command is used it will display-

* The current branch name
* If it is an initial commit or not
* List of the unstage/untracked files(which are newly added or modified but not added to staging/index) List of the files pending to commit(these files are previously added to stating/index)

If there is no file/folder to add or commit then it will just display name of the branch and message "nothing to commit, working directory clean"

**what is the different ways to add/stage files/changes to the git repository?**

There are two ways to add or stage the files or the changes to the git repository.

Using 'git add' command, this will add the file to staging

Using 'git commit -am <your custom message>', the '-a' option with commit command will add the files to staging(not to 'newly added files' but only that have been modified and deleted) and commit too. basically 2 in 1, it adds then commit the files. '-m' options for providing the custom commit message.

**What are the steps to unstage files and discard the charges in working directory?**

If the git status command displays 'myfile.txt' as file pending to commit then

1. To unstage a file(which is not yet committed but in staging stage); command 'git reset <file-name>' is used. After using 'git reset HEAD myfile.txt' command; 'git status' command will not show myfile.txt in the list of files pending to commit, now it will be displayed under the list of the unstage/untracked files.
2. Now to discard the changes('myfile.txt') in working directory; command 'git checkout -- <file-name> is used. After using 'git checkout -- myfile.txt' command; 'git status' command will not show myfile.txt in the list of unstage/untracked files as this file 'myfile.txt' will be replaced with the last committed version from the repository.

**What is the git stash?**

When you are working in your working directory and done few changes so few items in staging, and now if you have to provide any hot fix based on the last commit on the remote repository then to do it the best way is save the current state of the working directory and the index/staging (something like on a stack), for that we use command 'git stash' or 'git stash save'. After using this command, the current working directory will be having the same match as on the remote last commit means locally no changes exist to stage or commit. Now you can do your stuff on the clean copy of the remote; git stash will not cover git ignored files and the newly created files but not 'staged' yet in current working directory.

**What is the difference between "Git Pull" and "Git fetch"?**

Pull

$ git pull origin master

git pull says "**bring the changes in the remote repository to where I keep my own code.**"

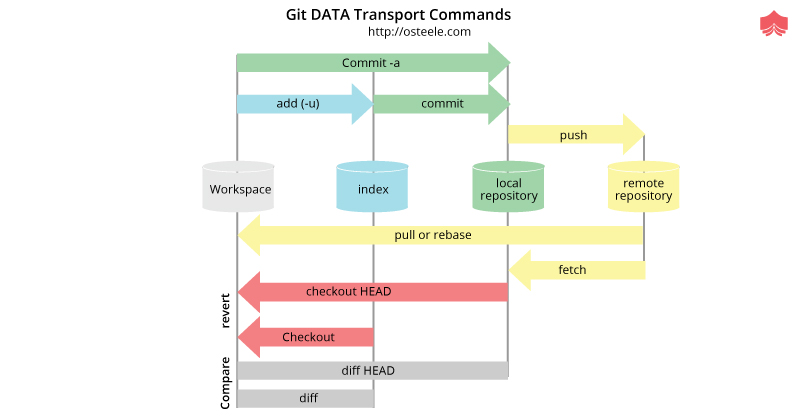
Normally git pull does this by doing a git fetch to bring the local copy of the remote repository up to date, and then merging the changes into your own code repository and possibly your working copy. If you don’t closely manage your branches, you may run into frequent conflicts.

Fetch

$ git fetch origin

git fetch is the command that says "**bring my local copy of the remote repository up to date.**"

When you command fetch, Git extract all commits from the target branch that does not exist in the current branch and keeps them in your local repo. But, it does not merge them with your current branch. This is mostly useful if you need to keep your repo up to date, but are working on some solution that might break if you update your files. To integrate the commits into your master branch, we use merge.



**How to revert the previous commit in git?**

git reset --hard HEAD~N- To undo last commit and changes.

N stands for how far you would like to undo your changes. For undo last changes N=1

Let's talk about below use case where C is current HEAD while F is a state of files.

  (F)

A-B-C

   ↑

 master

If you like to undo commit C. You do this:

git reset --hard HEAD~1

The result is:

(F)

A-B

 ↑

master

Now B is the HEAD. Because you used --hard, your files are reset to their state at commit B.

Suppose we would like to undothe commit but keep our changes before we do good commit. Let's start again with the above example, with C as our HEAD:

  (F)

A-B-C

   ↑

 master

We can easily achieve this by, leaving off the --hard:

git reset HEAD~1

In this case, the result will be below :

  (F)

A-B-C

 ↑

master

In both scenarios, the HEAD was just a pointer to the last commit. When we run a git reset HEAD~1, we ask Git to move the HEAD pointer one commit back. But (unless we use --hard) we leave our files as they were. So now git status will show the changes you had checked into C. Your files will remain as it is but only GIT HEAD has moved one commit back.

**What is the difference between Head woking tree and index, in Git?**

A single git repository can track any number of branches, but your working tree is always referring just one of them (the "current" or "checked out" branch), and HEAD points to that branch.

HEAD is the commit at the top of the current branch. If you've just checked out the branch, i.e. have no modified files, then its content matches the working tree. As soon as you modify anything, it no longer matches.

A working tree is the dir and sub-directory within it that contain the source files. It can be anywhere, but normally it is the same dir in which the hidden .git dir is located.

The index/staging area is a single, large, binary file in <baseOfRepo>/.git/index, which lists all files in the current branch, their sha1 checksums, time stamps and the file name - it is not another directory with a copy of files in it.

**How to remove a file from git without removing it from your file system?**

Sometimes we may end up adding files in our commit which we never intended to commit. Nothing to worry if you come across a scenario. The command git rm will remove it from both your staging area (index), as well as your file system (working tree), which may not be what you want.

We can use git rm --cached on the file if you want to remove from the version control cache but do not want to remove/delete from your filesystem. So if you wanted to remove foo.txt from version control like this just run this command:

git rm --cached foo.txt

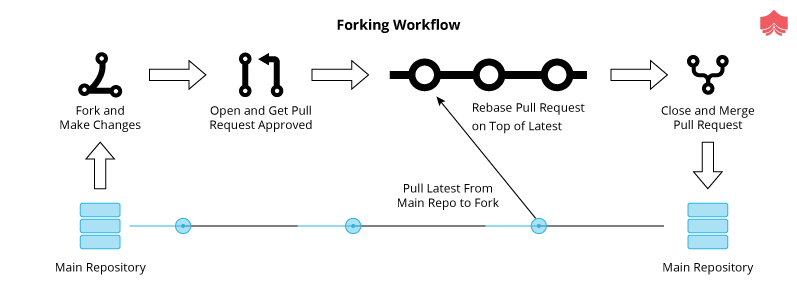
We can also use below git commands to achieve the same result:

git reset filename

The above command is used to undo the local changes. Git reset operates on "**The Three Trees of Git**". These trees are the Commit History ( HEAD ), the Staging Index, and the Working Directory

**Explain the advantages of forking workflow?**

The Forking Workflow is fundamentally different than other popular Git workflows. Instead of using a single server-side repository to act as the “central” codebase, it gives every developer their own server-side repository. The Forking Workflow is most often seen in public open source projects.



The forking workflow can be summarized as below:

* Fork a GitHub repository.
* Clone the forked repository to your local system.
* Add a Git remote for the original repository.
* Create a feature branch in which to place your changes.
* Make your changes to the new branch.
* Commit the changes to the branch.
* Push the branch to GitHub.
* Open a pull request from the new branch to the original repo.
* Clean up after your pull request is merged.

**what is the "git cherry-pick"?**

Cherry picking in Git shifts a commit from one branch and apply it onto another one. This is indifferent to other ways like merge and re-bases which moves many commits onto another branch. Make sure we are on the same branch we would like to apply the commit to.

1. We will first move to master branch  
   git checkout master
2. Once we are on the same branch we run the following command:  
   git cherry-pick <commit-hash>

**Note**:commit-hash: Unique identifier for  commit.

If we cherry-pick from a public repo branch, we should consider below command :

git cherry-pick -x <commit-hash>

This will always generate a standardized commit message. This way, our team members can still keep track of the origin of the commit and may avoid merge conflicts in the future. If we have notes attached to the commit that does not follow the cherry-pick. To bring them over as well, we can use:

git notes copy <from> <to>

A common use case for cherry-picking is to forward- or back-port commits from a maintenance branch to a development branch.

**What is the git fork? what is the difference between fork, branch and clone?**

Clone: By cloning a repository, you download the whole repository with dotfiles (like .git, .gitignore etc.) to your local computer. A clone is a localized copy of some remote repository. When we clone, we are actually bringing the entire source repository, including all the history and branches locally.

Fork: By forking a repository, you just virtually create a “copy” of the main repository to your account. Forks are usually useful in creating Pull Request(s) to the main repository. A fork is a remote, server-side copy of a repository, distinct from the original.

A branch is a way to handle the changes within a single repo to eventually integrate them with the rest of the code. A branch exists within a repository only. Conceptually, it represents a thread of development which is needed to manage development/maintenance.

The owner of the main repository gets a notification on their newsfeed when you fork their repository, however, it’s impossible to track if you clone / download their repository. However, note that you won’t be able to submit Pull Request(s) with a download as there are no dotfiles

**what language is used in git ?**

GIT is fast, and ‘C’ language makes this possible by reducing the overhead of runtimes associated with higher languages.

Git’s Source code is hosted on Github here: [git/git](https://github.com/git/git)

So, From that these are the languages used in that repository:

* C - 45%
* Shell - 35%
* Perl - 8%
* Tcl - 5%
* Python - 2%
* C ++ - 2%

Git's design was inspired by BitKeeper and Monotone. Git was originally designed as a low-level version-control system engine, on top of which others could write front ends, such as Cogito or StGIT. The core Git project has since become a complete version-control system that is usable directly. While strongly influenced by BitKeeper, Torvalds deliberately avoided conventional approaches, leading to a unique design

The below are characteristics of GIT which makes it very popular in SCM category:

1. Strong support for non-linear development.
2. Compatibility with existing systems and protocols
3. Efficient handling of large projects
4. Cryptographic authentication of history
5. Toolkit-based design.

**What is the function of 'Git push' in Git?**

‘GIT PUSH’ updates remote refs along with associated objects. The "**push**" command is widely used to publish any new local commits on a remote server. The source (i.e. which branch the data should be uploaded from) is always the currently checked out HEAD branch. The target (i.e. which branch the data should be uploaded to) can be specified in the command's option. Sometimes we use below command :

**git push origin master**

**push** = push your changes to the remote server

**origin** = remote Server origin

**master** = Master branch

If you have other remote branches you have something like "git push origin test" then you push your changes to the test remote branch.

Just specifying

git push origin

will push every local branch that has a matching remote branch to that branch per default. Not just the current branch. This is the same as using git push origin:



**What is the difference between git checkout and git clone?**

git clone is to fetch your repositories from the remote git server.

we clone a remote  repository with below command

**git clone [url]**

For example, Let’s say  if we want to clone the Open Framework Git library called open\_framework, we can easily do so like this:

**$ git clone git://github.com/SU-SWS/open\_framework.git**

That will create a directory named open\_framework (at our local file system location), and initializes a .git directory inside it, extract all the data for that repository, and checks out a working copy of the latest version. If we go into the newly created open\_framework directory, we can see the project files in there, ready to be worked on or used.

Cloning a Repository Into a Specific Local Folder

If we want to clone the repo into a particular directory named something other than open\_framework, we can specify that as the next command-line option:

**$ git clone git:github.com/SU-SWS/open\_framework.git mynewtheme**

The above command does the same thing as the previous one, but the clone gets created in the target directory called mynewtheme.

git checkout is to check out the desired status of your repository (like branches or particular files).

E.g., you are currently on the master branch and you want to switch into the develop branch.

git checkout develop\_branch

**what is the difference between git and svn?**

GIT is a distributed version control system where each developer gets its own local copy of the remote repository. While SVN comes under centralized version control. Centralized implies a client-server relationship. There is one main repository (repo) on a server, and every working copy clones and commits to this server. Decentralized is the opposite: there is no “**central**” repo.

You can not commit in SVN if you are not connected to the central repo so network connectivity is mandatory for SVN for any commit. With Git, you do not have this issue. Your local copy is a repository, and you can commit to it and get all the benefits of source control. When you regain connectivity to the main repository, you can commit against it.

| **Git** | **SVN** |
| --- | --- |
| 1. Decentralized Version Control tool | 1. Centralized Version Control tool |
| 2. It comes under the third generation of version control tools | 2. It comes under the 2nd generation of Version Control tools. |
| 3. We can clone the entire repo on our local systems | 3. Version history is stored on a server-side repository |
| 4. The commits are possible even if remote repo not available | 4. Only online commits are allowed |
| 5. The basic /pull operations are faster | 5. The Push/pull operations are slower as compared to GIT. |
| 6. Works are shared automatically by commit | 6. Nothing is shared automatically, you need manual intervention |

**What is 'head' in git and how many heads can be created in a repository?**

In every repository, there is a default head referred to as “Master”. When you switch branches with git checkout, the HEAD revision changes to point to the tip of the new branch. A repository can contain any number of heads. While HEAD may regularly be portrayed as indicating the latest submit or indicating what branch you're on, neither one of the descriptions is very right.

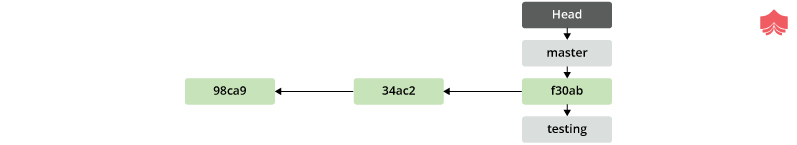
It focuses on what you've at present looked at, pretty much, which may not be what's the most recent. HEAD is a common content record in .git/HEAD, so you can investigate. So at this moment, my HEAD contains:

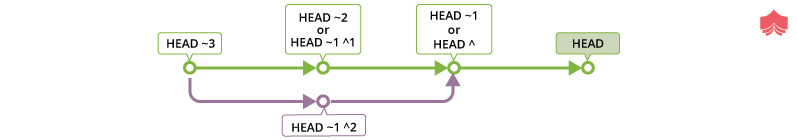
ref: refs/heads/test

This implies I've looked at the 'test 'branch, which is indicated by the ref 'refs/heads/test'. That this is (usually\*) put away in the record .git/refs/heads/test, and contains:

7fc4bd269202ead49b9e5d4fa5f8e7db1525ea5b

which is the commit ID for that branch.





~(Tilde) and ^(caret), pointing to the relative position of a commit. The most common is to add ~(Tilde) after the HEAD to point to the generation of the parent; ^(caret) can point to the previous generation of generations.

**What git repository hosting services is widely used?**

A source-code storehouse is a chronicling and web facilitating office where source code, for programming or for site pages, is kept, either openly or secretly. They are frequently utilized by open-source programming ventures and other multi-engineer activities to deal with different forms.

Kindly find beneath probably the most well known facilitating administrations accessible :

**Alternative 1: GitHub**

It's as yet the biggest network site for programming advancement, despite everything it has probably the best apparatuses for issue following, code audit, persistent reconciliation, and general code the board. the universally adored open source appropriated rendition control framework.

**Choice 2: GitLab**

It's completely open source. You can have your code directly on GitLab's site much like you would on GitHub, yet you can likewise decide to self-have your very own GitLab case individually server and have full power over who approaches everything there and how things are overseen. GitLab practically has highlighted equality with GitHub, and a few people may even say its ceaseless incorporation and testing devices are prevalent. In spite of the fact that the network of engineers on GitLab is unquestionably littler than the one on GitHub,

**Choice 3: Bitbucket**

Bitbucket has been around for a long time. Bitbucket was procured by a bigger partnership (Atlassian) eight years back It's as yet a business stage like GitHub, however, it's a long way from being a startup, and it's on really stable balance, hierarchically. Bitbucket shares the vast majority of the highlights accessible on GitHub and GitLab, in addition to a couple of novel highlights of its own, similar to local help for Mercurial storehouses.

**Alternative 4: SourceForge**

The granddaddy of open source code store locales is SourceForge. It used to be that on the off chance that you had an open source venture, SourceForge was the spot to have your code and offer your discharges. It took a short time to move to Git for adaptation control, and it had its very own rash of the business obtaining and re-securing occasions, combined with a couple of tragic packaging choices for a couple of open source ventures. So, SourceForge appears to have recuperated from that point forward, and the site is as yet a spot where many open source extends live.

<https://www.git-tower.com/blog/git-hosting-services-compared/>

**what are the advantages of using git ?**

GIT is a distributed version control system where each developer gets its own local copy of the remote repository. Having a full nearby history makes Git quick since it implies you needn't bother with a system transfer speed to make submits, examine past renditions of a record, or perform diffs between submits. Conveyed advancement likewise makes it simpler to scale your group. If somebody breaks the creation branch in SVN, different designers can't check in their progressions until it's fixed. With Git, this sort of circumstance doesn't exist. Everyone can proceed with their advancement in their very own neighborhood storehouses.

Some of the benefits of GIT are listed below :

a) Data redundancy and replication  
b) High availability  
c) Only one.git directory per repository  
d) Superior disk utilization and network performance  
e) Collaboration friendly

**What is the git rebase and how can it be used to resolve conflicts in a feature branch before the merge?**

In simple words, *git rebase* allows one to move the first commit of a branch to a new starting location. For example, if a feature branch created from the master branch, and in between if the master branch has received additional commits, *git rebase* can be used to move the feature branch to the tip of master. The command will integrate the changes made in the feature branch at the tip of the master, allowing conflicts to be resolved amicably. When it is done with care, this will always allow the feature branch to be merged into master seamlessly and sometimes as a simple fast-forward operation.

Rebase compresses all the changes into a single “patch.” Then it integrates the patch onto the target branch. Unlike merging, rebasing flattens the history because it transfers the completed work from one branch to another.you should never rebase commits once they've been pushed to a public repository. The rebase would replace the old commits with new ones. We can have a scenario where if we started doing some development and then another developer made an unrelated change. You probably want to pull and then rebase to base your changes from the current version from the repo.

**when should we use Git stash?**

This command saves our local modifications and reverts the working directory to match the HEAD commit of the repository. The changes stashed away by above command can be listed with git stash list, and checked with git stash show, and restored with git stash apply. We can understand this scenario with the below examples. In the below code we have modified one file (index.html) and added new file stsyle.css in the repository.

$ git status

On branch master

Changes to be committed:

new file: style.css

Changes are not staged for commit:

modified: index.html

$ git stash

Saved working directory and index state WIP on master: 5002d47 our new homepage

HEAD is now at 5002d47 our new homepage

$ git status

On branch master

nothing to commit, working tree clean

There are lots of scenarios where we would be needing a clean working copy as recommended or even required: This situation arises when merging branches, when pulling from a remote, or simply when checking out a different branch.

The "git stash" command always help us to (temporarily but safely) and store our uncommitted local changes - and leave us with a clean working copy.

Continuing Where You Left Off

As already mentioned, Git's Stash is meant as temporary storage. When you're ready to continue where you left off, you can restore the saved state easily:

$ git stash pop

**What is the function of 'Git Config'?**

Git has an in-built command called git config that gives you a chance to get and set setup factors that control all parts of what Git looks like and works in your repository. The 'git config' order is an advantageous method to set design alternatives for your Git installation. The behaviour of a store, client data, inclinations and so forth can be characterized through this direction. The primary thing you ought to do when you introduce Git is to set your client name and email address. This is significant in light of the fact that each Git submit utilizes this data, and it's changelessly prepared into the submits you begin making:

Setting your Git username for *every* repository on your computer

1. Open Git Bash.
2. Set a Git username:

    $ git config --global user.name "*XYZ*"

1. Confirm that you have set the Git username correctly:
2. $ git config --global user.name  > XYZ

**Checking Your Settings**

If you want to check your configuration settings, you can use the git config --list command to list all the settings Git can find at that point:

$ git config --list

user.name=XYZ

user.email=abc@example.com

color.status=auto

color.branch=auto

color.interactive=auto

color.diff=auto

**What is the function of "Git diff" in git?**

'git diff ' dependably show the progressions between various submits, submit and working tree etc. Git has an Inuit Index between local repo and our working directory. So the vast majority of Git directions really allude to list or the local repo. When we state HEAD in our Git direction, it really alludes the local repo.

//contrast the working directory and local repo.

**git diff HEAD [filename]**

//contrast the working directory and index.

**git diff [filename]**

//contrast the index and local repo.

**git diff - cached[filename]**

We can likewise look at records from two distinctive submits. Each submit in Git has a hash submit id which we can get when we type git log. At that point, we can utilize the submit id with diff order this way.

**git diff 8ez2..e13 922...a3f86**

We can look at a solitary file, yet the entirety of our progressions without a moment's delay. On the off chance that we have made changes in numerous files, simply don't make reference to any file name in the diff direction which will diff all the changed records.

//contrasts working directory and index,

//for example, demonstrates the progressions that are not staged yet.

**git diff**

//contrasts working catalog and local repo

//demonstrates the rundown of changes after your last submit.

**git diff HEAD**

//contrasts index and local repo

//demonstrates the diff between your last submit and changes to be submitted straightaway.

**git diff - cached**

**What is the "git status" is used for?**

A 'Git Status' is the most generally utilized command on GIT platform. It records down the distinction between the working catalog and the index, it is useful in understanding a git all the more completely.

To Check the status of the local repo

Utilize the git status command, to check the present condition of the local repo.

$ git status

On branch master

Your branch is up-to-date with 'origin/master'.

nothing to commit, working directory clean

Being "fully informed regarding 'origin/master' is the best message for git status. It really implies there is nothing to push. "working registry clean" implies that every one of the documents in the present index is being overseen by git (or are as a rule deliberately overlooked by means of .gitignore) and the latest adaptation of the record has been submitted and nothing pending.

In the event that git status has referenced "Unmanaged records:", we may need to include at least one unmanaged file.

$ git status

On branch master

Your branch is ahead of 'origin/master' by 1 commit.

  (use "git push" to publish your local commits)

Untracked files:

  (use "git add <file>..." to include in what will be committed)

xyz.txt

nothing added to commit but untracked files present (use "git add" to track).

We will use the git status, to keep monitoring the states of both the working directory and the repository.

**How to resolve conflicts in git?**

Suppose you are going to pull most recent from storehouse and run over clash with neighborhood duplicate. We should attempt to comprehend this situation with underneath precedents :

git pull origin master ( Trying to get the latest from remote repo)

From giturl/projectname

\* branch            master -> FETCH\_HEAD

Updating b044a4a..gg36324

error: Entry abc' not uptodate. Cannot merge.

A plainly specific file(abc) isn't up to date. Now you will attempt to set your nearby duplicates right by running beneath commands:

git add abc

git commit -m "added files "

git pull origin master

From  giturl/projectname

\* branch            master -> FETCH\_HEAD

Auto-merging abc

CONFLICT (content): Merge conflict in abc

Automatic merge failed; fix conflicts and then commit the result.

So you decide to take a look at the changes by running mergetool :

git mergetool

Then you realize that you would like to keep your copies rather than a remote copy as the remote copy is not up to date :

git checkout --ours abc

git checkout --theirs abc

git add abc

git commit -m "using theirs"

And then we try a final time

git pull origin master

From  giturl/projectname

\* branch            master -> FETCH\_HEAD

Already up-to-date.

**List some useful commands in git?**

There are some helpful GIT commands which are generally utilized amid advancement. A portion of the essential GIT commands recorded down underneath classified into various categories:

git init Initialize a nearby Git store

git clone remoteurl/[username]/[repository-name].git Create a nearby duplicate of a remote store

git branch List branches (the reference mark indicates the present branch)

git branch - a List all branches (nearby and remote)

git branch [branch name] Create another branch

git branch - d [branch name] Delete a branch

git status Check status

git add [file-name.txt] Add a document to the arranging zone

git add- A Add all new and changed records to the organizing region

git commit- m "[commit message]" Commit changes

git rm - r [file-name.txt] Remove a document (or organizer)

git push origin- delete[branch name] Delete a remote branch

git checkout - b [branch name] Create another branch and change to it

git checkout - b [branch name] origin/[branch name] Clone a remote branch and change to it

git checkout [branch name] Switch to a branch

git checkout - Switch to the branch last looked at

git checkout - [file-name.txt] Discard changes to a record

git merge[branch name] Merge a branch into the dynamic branch

git  merge[source branch] [target branch] Merge a branch into an objective branch

git stash Stash changes in a filthy working catalog

git stash clear Remove all reserved sections

git push origin [branch name] Push a branch to your remote store

git push - u origin[branch name] Push changes to remote storehouse (and recall the branch)

git push Push changes to remote storehouse (recollected branch)

git push origin- delete[branch name] Delete a remote branch

git pull Update nearby archive to the most up to date submit

git pull cause [branch name] Pull changes from remote archive

git remote add origin remoteurl/[username]/[repository-name].git Add a remote storehouse

git remote set-url origin remoteurl/[username]/[repository-name].git Set a repo birthplace branch to SSH

git log View changes

git log - summary View changes (nitty gritty)

git diff [source branch] [target branch] Preview changes before blending

**what are the steps of git local workflow?**

1. For starting local, we initialize our current working project directory using 'git init' or 'git init <your-project-name>'(this will create a new directory with provided your-project-name as working directory) command and on the GitHub site set up your GitHub repository.
2. Now in the local working directory, we do changes (either adding/editing files etc)
3. Then we're going to 'stage' our changes using the "git add" command
4. Then commit our changes that are in the staging area using the 'git commit -m <your custom message here>'
5. Setup a remote repository (origin) using command 'git remote add origin git@github.com:User/UserRepo.git' (you can change it later using command 'git remote set-url origin git@github.com:User/UserRepo.git')
6. Once we're ready to collaborate with others on the main repository, we'll push our changes up to our remote repository on GitHub(or the hosted remote repository) using command 'git push -u origin master'; here -u is upstream(use it only for first push command just once), origin is remote name and master is the branch's name.
7. Once everyone else had pushed their changes to the remote repository,then we'll do a pull from the remote repository to our local git repository using 'git pull' or 'git pull <remote-name> <branch-name>' command
8. Later on whenever doing new working the local repo; do 'git pull' and then start new work.

**what are the steps of git remote workflow?**

For example using GitHub as a remote repository

1. On the GitHub set up your remote GitHub repository
2. Then do a git's clone using 'git clone' command to create a new repository on our local system.
3. Now in the local working directory we do changes(either adding/editing files etc)
4. Then we're going to 'stage' our changes using the "git add" command
5. Then commit your changes that are in the staging area using the 'git commit -m <your custom message here>'
6. Setup a remote repository(origin) using command 'git remote add origin git@github.com:User/UserRepo.git' (you can change it later using command 'git remote set-url origin git@github.com:User/UserRepo.git')
7. Once we are ready to collaborate with others on the main repository, we'll push our changes up to our remote repository on GitHub(or the hosted remote repository) using command 'git push -u origin master'; here -u is upstream(use it only for first push command just once), origin is remote name and master is the branch's name.
8. One everyone else had pushed their changes to the remote repository,then we'll do a pull from remote repository to our local git repository using 'git pull' or 'git pull <remote-name> <branch-name>' command
9. Later on whenever doing new working the local repo; do 'git pull' and then start new work.

**How to do git configuration to get or set various user options?**

When reading/getting, the values are read from the system, global and repository local configuration files by default, and options --system, --global, --local and --file <filename> can be used to tell the command to read from only that specific location. When writing, the new value is written to the repository local configuration file by default(--local), and options --system, --global, --file <filename> can be used to tell the command to write to that location.

Set a user's email-id (to be recorded in any newly created commits) : git commit --global user.email "sc.2017.india@gmail.com"

Set a user's full name (to be recorded in any newly created commits) : git commit --global user.name "Sandeep Choudhary"

Set signed key for a signed tag or commit : git config --global user.signingkey <your-signed-key-here> Command 'git config --global --list' will show all of the current global settings/options.

Configuration saved using '--global' is stored in a file '.gitconfig' in the current user's home directory. User's name and email-id should be set before using git further.

**How to view the commit history of the repository?**

Using command ‘git log"; this command displays commits list(in chronological order) from latest to old one in descending order of the committed date. For each commit record it displays-

Commit id (SHA-1 has code)

Author's (user's) name and email-id Date and time

Commit message provided at time of commit

To make the output more compact or custom; we can use lot of other options along with command e.g. 'git log -- oneline'

**what are the steps to permanently remove a committed file from the repository?**

To remove a file for example 'myfile.txt'; which is already committed and a part of repository, the first step is to 'staging' this file deletion using 'git rm myfile.txt' command. After this command if we use 'git status' it will show message e.g. 'deleted: myfile.txt' . Now as second step to make it permanently we have to commit this staging using 'git commit -m <your message>' command.

Another way is if we remove the file 'myfile.txt' from the working directory manually for example using terminal rm command or using menu as we delete file normally using operating system's GUI. post manual removal the 'git status' command will display the message 'Changes not staged for commit:' and file information like 'deleted: myfile.txt'. To stage these manual changes(we can say same behaviour as working directory changes of add/remove/update files) we have to use -u (recursively update) option with add e.g. 'git add -u'(pre git 2.0 version) or 'git add .'(git 2.0 and above version) command. Then use git commit command to make them permanent.

**How do you restructure the working directory file in the git repository?**

For example if we have a file called 'myfile.txt' to restructure/move in the sub directory 'mydir' under git's working directory; rather than cut and paste(manual mv command), we can use 'git mv <file-name> <target-dir-name>' e.g. 'git mv myfile.txt mydir'. This will move the file to provided directory and marks it to 'staging'. If we use 'git status' command now, it will display changes available to commit e.g. 'renamed: myfile.txt -> mydir/myfile.txt'. Now using git commit command make this restructuring permanently in the repository.

If we move file/folder out of the working directory (basically like permanently removing a committed file from the repository) then post movement we have to use 'git add -u' command to mark it staging and then commit command to make it permanently.

**How to configure git to not to add and commit non-source file like IDE configuration of debug log files?**

In the working directory, create a new file '.gitignore'; basically this file's each line will contain a entry(file or folder name or file extension with filter) to be ignored by git to add or commit in the repository. for example if we want ignore all of the file with '.debug' extension then add a entry '\*.debug' in the file. Stage this file '.gitignore' using 'git add .' and then commit it to repository.

**How do you setup and use SSH authentication to connect with github/remote repository?**

1. To do SSH setup for git; we need to create a '.ssh' directory in the user's home directory.
2. Then in inside (using 'cd' command) this (e.g. '/Users/SandeepChoudhary/.ssh') newly directory, use command 'ssh-keygen -t rsa -C "<your-email-id>"'. Here -t is used to specify type of key to create and -C(in upper case only) to provide the comment in this case the email-id.
3. Then during execution of this command, provide inputs like name of file to save the key(just press enter so by default it will be saved in 'id\_rsa' file, provide passphrase. Now a file 'id\_rsa.pub' will be created in '.ssh. Directory.
4. Now on the github under 'Account Settings' > 'SSH keys' > 'Add SSH key', provide custom title as this ssh key will be tightly coupled with your specific machine where ssh key is generated. Under key option, copy paste the content of the 'id\_rsa.pub' file from your machine to here and click 'add key'.
5. Now on your machine using command 'ssh -T git@github.com' you will establish a ssh connection/tunnel with github.

**How do you manage set of tracked/remote repository?**

We add a remote repository(origin) using command 'git remote add origin git@github.com:User/UserRepo.git' (you can change it later using command 'git remote set-url origin git@github.com:User/UserRepo.git').

We can list the remote repositories using 'git remote -v' command and this will display the name and the url of the remote repositories.

We push our changes up to our remote repository using command 'git push -u origin master'; here -u is upstream(use -u only for first push command just once), origin is remote name and master is the branch's name.

Then we'll do a pull from remote repository to our local git repository using 'git pull' or 'git pull <remote-name> <branch-name>' command to receive all your remote changes (commits) from the remote repository.

**How to know what are the different saved states/stash of the current working directory?**

Using the command 'git stash list'. The output of this command is a list of the saves states starting from zero the latest to the oldest one. Each entry has the stash index starting from zero e.g. stash@{n} where n is index, followed by the branch name and the summarized description.

what is git stash drop?

If at any point of time we want to remove a single stashed state from the stash list, we use the command 'git stash drop'(it will remove the latest from list i.e. stash@{0}) or 'git stash drop stash@{n}'; here stash@{n} must be a valid stash log reference as displayed in stash list command's output.

**Explain the steps to work on any production issue in git ?**

Let's go through the steps to understand flow in case we are working on any production issue :

**Step 1**: We will create a local branch for developing any feature /bug which will be a child branch inherited from the production branch. The below command will create headerissuebranch locally on which developer would work and fix production issues.

git checkout -b headerissue production branch

We can check the branches existing locally by running below command.

**step 2**: git branch -List all branches existing locally

You can progress the changes by running below command

**step 3**:git status- To see any changes in the branch

**step 4**:Once you are done with changes committing changes locally not remotely.  Once unit testing and Release testing gets completed, you can move to merge the changes in the production branch.

**step 5**:  Now you can move to the production branch

    $ git checkout productionbranch

Switched to branch productionbranch and merge the changes once

   $ git merge --no-ff headerissue (merging with issue branch)

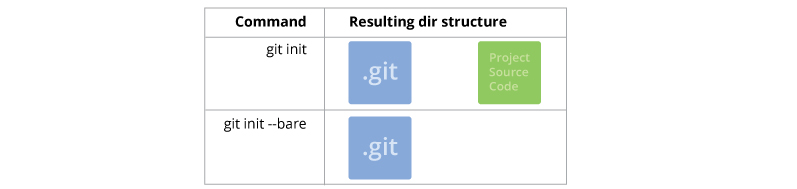
   $ git branch -d headerissue(delete local branch)

   $ git push origin productionbranch (pushing issue fix in production branch)

**what do understand by 'Bare repository' in git ?**

A git bare repository is helpful when you would like to collaborate with other users but don’t want to use a hosting service like GitHub. In such cases, you can set up a remote server that is only accessible by your internal team, create a bare repository over there, which acts as the central git server from which each of you can push and pull your code from your local boxes.

A bare repository made with git init -- bare is for sharing. On the off chance that you are working together with a group of designers, and need a spot to share changes to a repo, at that point you will need to make a bare repo in a centralized spot where all clients can push their changes (frequently the simple decision is GitHub). Since GIT is an appropriated rendition control framework, nobody will legitimately alter documents in the common brought together storehouse. Rather designers will clone the mutual bare repo, make changes locally in their working duplicates of the repo, at that point push back to the commonly exposed repo to roll out their improvements accessible to different clients.



Since nobody ever makes alters straightforwardly to records in the mutual bare repo, a working tree isn't required. Actually, the working tree would simply get in the way and cause clashes as clients to push code to the store. This is the reason uncovered storehouses exist and have no working tree.

How do I make existing non-bare repository bare?

After making sure that there are no uncommitted changes, etc.:

$ mv repo/.git repo.git

   $ git --git-dir=repo.git config core.bare true

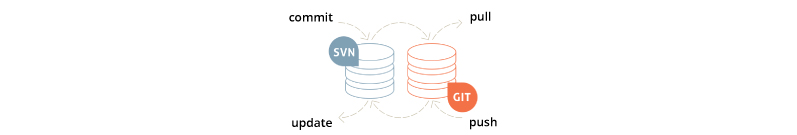
   $ rm -rf repo

**what is a sub git?**

SubGit is a device for empowering a consistent and calm SVN to Git relocation. It makes a writable Git reflection of a neighbourhood or remote Subversion archive and utilizes both Subversion and Git as long as you can imagine. It creates a bi-directional mirror that can be used for pushing to Git as well as committing to Subversion. SubGit also takes care of synchronization between Git and Subversion. It requires access to your GitLab server as it interacts with the Git repositories directly in a filesystem level.

Some of the benefits of using SubGit tool are as follows:

1. Single, concentrated mirror area and set-up
2. No uncommon directions - utilize the full intensity of Git on the customer side
3. Move clients and instruments to Git at agreeable pace with zero vacation and no medium-term switches
4. Handle standard, custom or single-catalogue SVN ventures



What is git hooks?

Sometimes there is a need to enforce policies, ensure consistency, and control your environment, and even handle deployment tasks. Git hooks are a basic idea that was actualized to address a need. When creating programming on a common undertaking, keeping up style direct norms, or while conveying programming (all are circumstances that git is regularly included with), there are frequently dull errands that you will need to do each time an activity is taken. Git snares are occasion based. When you run certain git directions, the product will check the hooks catalogue inside the git store to check whether there is related content to run. Git hooks are run locally.

Some example hook scripts include:

* **pre-commit**: Check the commit message for spelling errors.
* **pre-receive**: Enforce project coding standards.
* **post-commit**: Email/SMS team members of a new commit.
* **post-receive**: Push the code to production.

Every Git repository has a .git/hooks folder with a script for each hook you can bind to. You're free to change or update these scripts as necessary, and Git will execute them when those events occur.

Git hooks are divided into two categories:

* **Client-Side Hooks:**Hooks that are called and executed on the committer's computer.
* **Server-Side Hooks:** These hooks are executed on servers that are used to receive pushes.

**Why do we use Git bisect?**

GIT cut up completes a paired inquiry to locate a specific relapse which is a guilty party of submitting some wrong code in archive. It is preposterous to expect to assess every single resolve to discover the relapse as it is very time-consuming. Suppose you have beneath improvement history :

... - 0 - 1 - 2 - 3 - 4\* - 5 - current

You come to realize that your program isn't working accurately at the present correction, and it was working at the modification 0. So the relapse was likely presented in one of the submits 1, 2, 3, 4, 5, current.

You can attempt to check each submits, manufacture it, check if the relapse is available or not. On the off chance that there is countless, this may take quite a while. This is a straight pursuit. We can improve by completing a double pursuit. This is the thing that the git divide order does. At each progression, it attempts to decrease the number of amendments that are conceivably awful significantly.

You'll utilize the order this way:

$ git stash save

$ git bisect start

$ git bisect bad

$ git bisect good 0

Bisecting: 2 revisions left to test after this (roughly 2 steps)

[< ... sha ... >] 3

After this direction, git will check out a submit. For our situation, it'll be submitting  3. You have to fabricate your program and check whether the relapse is available. You'll likewise need to tell git the status of this modification with either git bisect awful if the relapse is available, or git bisects great on the off chance that it isn't.

How about we guess that the relapse was presented in submit 4. At that point, the relapse is absent in this update, and we tell it to git.

$ make

$ make test

... ... ...

$ git bisect good

Bisecting: 0 revisions left to test after this (roughly 1 step)

[< ... sha ... >] 5

It will then check out another submit. Either 4 or 5 (as there are just two submits). We should assume it picked 5. After manufacture, we test the program and see that the relapse is available. We at that point tell it to git:

$ make

$ make test

... ... ...

$ git bisect bad

Bisecting: 0 revisions left to test after this (roughly 0 steps)

[< ... sha ... >] 4

We test the last modification, 4. Furthermore, since the one presented the relapse, we tell it to git::

$ make

$ make test

... ... ...

$ git bisect bad

< ... sha ... > is the first bad commit

< ... commit message ... >

In this basic circumstance, we just needed to test 3 forms (3, 4, 5) rather than 4 (1, 2, 3, 4). This is a little win, however, this is on the grounds that our history is so little. On the off chance that the pursuit run is of N submits, we ought to hope to test 1 + log2 N submits with git bisect rather than generally N/2 submits with a direct hunt.

When you've discovered the submit that presented the relapse, you can ponder it to discover the issue. When this is done, you use git bisect reset to return everything on the first state before utilizing git bisect direction.

**What are branching strategy in git?**

it is important to understand the basics of Git version control. There are a parcel of individuals that utilizes

IDE's and visual instruments to do the essential tasks like commit and push changes, make and union

branches, think about history and return changes, without figuring out how Git really works. The

expanding technique relies upon the span of your group, the number of groups taking a shot at the

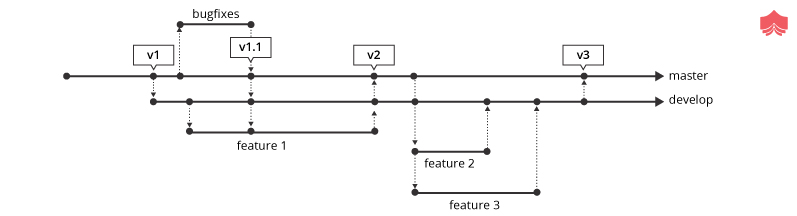
venture, if the undertaking has more than one element being created in the meantime, the recurrence

you discharge code to generation… the more unpredictable your situation, the more you depend on an

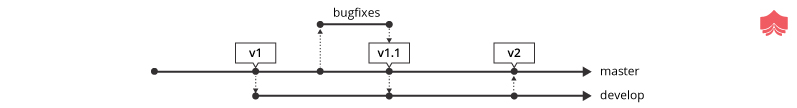
appropriate branch methodology to help it

There are three popular branching models in GIT.

* A component branch model keeps the majority of the progression for a specific element within a branch. At the point when the element is completely tried and approved via computerized tests, the branch is then converged into an ace.



* In this model, each undertaking is actualized without anyone else branch with the assignment key incorporated into the branch name. It is anything but difficult to see which code actualizes which task, simply search for the undertaking key in the branch name.



* Once the create branch has gained enough highlights for a discharge, you can clone that branch to shape a Release branch. Making this branch begins the following discharge cycle, so no new highlights can be included after this point, just bug fixes, documentation, and other discharge situated assignments ought to go in this branch. When it is prepared to dispatch, the discharge gets converged into an ace and labelled with a variant number. Moreover, it ought to be converted once again into creating a branch, which may have advanced since the discharge was started.

**One of your teammates accidentally deleted a branch and has already pushed the changes to the central git repo. there are no other git repos, and none of your other teammates had a local copy. How would you recover this branch?**

Check out the latest commit to this branch in the reflog, and then check it out as a new branch. Reflog is a mechanism to record when the tip of branches are updated. This command is to manage the information recorded in it. Basically every action you perform inside of Git where data is stored, you can find it inside of the reflog.

git reflog won't navigate HEAD's progressive system by any stretch of the imagination. The reflog speak to the requested rundown of the submits that HEAD has indicated: it's fixed history for our repo. The reflog isn't a piece of the repo itself (it's put away independently to the submits themselves) and is excluded in pushes, gets or clones; it's absolutely neighbourhood. you can't generally lose information from your repo once it's been dedicated. On the off chance that you incidentally reset to a more established submit, or rebase wrongly, or whatever other tasks that outwardly "evacuate" submit, you can utilize the reflog to see where you were previously and git reset - hard back to that ref to reestablish your past state

**How do you find the list of files changed in particular commit?**

Sometimes you require to inspect files which were part of a particular commit. Each commit is tagged with hash. You can use below command to see the list of files changed during particular commit.

**git diff-tree -r {hash}**



hash- commit Id

Given the submit hash, this will list down every one of the records that were changed or included that submit. The - r flag makes the order list singular records, instead of falling them into root index names as it were. The yield will likewise incorporate some additional data, which can be effectively stifled by including two or three flags::

**git diff-tree --no-commit-id --name-only -r {hash}**

https://d2o2utebsixu4k.cloudfront.net/media/images/1560418981401-image4.jpg

Here - no-commit- id will stifle the commit hashes from showing up in the output, and - name-only will just print the record names, rather than their ways.

**In how many ways we can refer commits in GIT?**

In Git, each commit is extraordinarily recognized by a novel hash number. These hashes can be utilized to distinguish the related commit much of the time, (for example, when we attempt to check out a specific condition of the code utilizing the git checkout {hash} order).

Aside from this, Git dependably keep data on various nom de plumes to guide to a specific commit, known as refs. Likewise, every single label that we make in the repo turns into a ref (and that is actually why we can utilize labels rather than submit hashes in different git directions). Git dependably keeps up various extraordinary false names that change depending on the condition of the archive, for example, HEAD, FETCH\_HEAD, MERGE\_HEAD, and so forth.

Git dependably permits resolves to be alluded as in respect to each other. For instance, HEAD~1 alludes to the commit parent to HEAD, HEAD~2 alludes to the grandparent of HEAD, etc. In the event of consolidation commits, where the commit has two guardians, ^ can be utilized to choose one of the two guardians, for example, HEAD^2 can be utilized to pursue the second parent.

Lastly, refspecs. These are constantly used to outline and remote branches together. In any case, these can be utilized to allude to commits that live on remote branches enabling one to control and control them from a neighborhood Git condition.